SAE TECHNICAL PAPER SERIES

2007-01-0783

Model-based Development for Event-driven Applications using MATLAB: Audio Playback Case Study

Peter J. Schubert
Packer Engineering, Inc.

Lev Vitkin
Delphi Electronics & Safety

David Braun
Purdue University

Reprinted From: Systems Engineering, 2007 (SP-2130)

ISBN 0-7680-1633-9

9 780768 016338

2007 World Congress
Detroit, Michigan
April 16-19, 2007

SAE International™

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-0790 Web: www.sae.org

All SAE papers, standards, and selected books are abstracted and indexed in the Global Mobility Database.

ISSN 0148-7191

Printed in USA

2007-01-0783

Model-based Development for Event-driven Applications using MATLAB: Audio Playback Case Study

Peter J. Schubert
Packer Engineering, Inc.

Lev Vitkin
Delphi Electronics & Safety

David Braun
Purdue University

Keywords: Executable Specifications, Systems Design, Model-Based Development, Audio Playback.

ABSTRACT

Audio playbacks are mechanisms which read data from a storage medium and produce commands and signals which an audio system turns into music. Playbacks are constantly changed to meet market demands, requiring that the control software be updated quickly and efficiently. This paper reviews a 12 month project using the MATLAB/Simulink/Stateflow environment for model-based development, system simulation, autocode generation, and hardware-in-the-loop (HIL) verification for playbacks which read music CDs or MP3 disks. Our team began with a "clean slate" approach to playback architecture, and demonstrated working units running production-ready code. This modular, layered architecture enables rapid development and verification of new playback mechanisms, thereby reducing the time needed to evaluate playback mechanisms and integrate into a complete infotainment system. A system simulation environment which included a real-time operating system supports generic mechanism and behavior models, to account for functional differences in playback mechanism from different suppliers. The execution of HIL system simulation required the design and implementation of a communication protocol device which transceives command messages passed between the system environment and the playback mechanism. With this harness, either a generic mechanism model or a hardware unit can be tested, allowing development of mechanism control software before production system hardware platforms are available. Specification-derived test vectors are used as functional tests during development and results have been used during the verification of auto-generated production software against the model. A comprehensive application of executable specs to audio playbacks is successfully demonstrated, including important considerations such as configuration management, model library management, and requirements traceability. Limitations and benefits of this approach are described, along with lessons learned in the implementation of model-based design in the field of automotive electronics.

CHALLENGES

Audio playback mechanisms ("playbacks") are typically the most complex building block in a radio. Aggressive price-cutting by playback vendors, and a steady flow of new functions and requirements all drive rapid change. Despite communication standards, playbacks vary in their implementation of IIC, or other protocols. On-board microprocessors within a playback may provide verbose error messages, just a modicum of information, or no feedback at all.

Automotive consumers demand interoperability of their music media. Despite the industry standard for Compact Disc Digital Audio (the "Red Book", IEC 908 – see CDDA trademark at right), many widely-available CD burning programs do not comply. Thus, while CD playback software developers strive for universality, there are many radio warranty complaints whose root cause is a non-compliant CD format.

Technology advances in consumer electronics are expected in automobiles more rapidly each year. From on-board vinyl record players pioneered by Delco Electronics (the forerunner of Delphi Electronics & Safety) in the 1950's, today's mobile audiophiles bring music stored on tape, on CDs (CDDA or MP3), on flash memory sticks, and on portable hard drives (e.g. iPod). In addition, vehicle manufacturers continue to bundle more features into car radios, such as navigation, safety warnings (seat belt reminder), Bluetooth® wireless communications, and satellite radio (e.g. XM™). These complex devices are no longer just a "radio", but are more accurately called "infotainment systems".

To handle the rapid change of such complex products, radio suppliers must accelerate software development, testing, and implementation for playbacks. Modular design within a universal architecture is a design goal, using customizable building blocks from a reuse library. The project described herein addresses these needs and successfully demonstrates the ability to rapidly create the control software to operate a new playback.

ARCHITECTURE

A modern radio is constructed of building blocks such as the AM/FM tuner, the human-machine interface (HMI), communications, the playback mechanism, and so forth. The various building blocks are run as tasks, controlled by the operating system (OS). The OS calls these tasks based on assigned priorities and inter-dependencies, triggered either internally, or in response to button-presses or media insertions on the radio faceplate. During and after the execution of its task, the playback module sends status information and error reports back to the OS. As a result, the OS will initiate the execution of other tasks, like displaying on the HMI, or sending communication messages to other tasks, such as volume control. Control of the sound profile (volume, equalizer, etc.) is performed on the radio main board (microprocessor, digital signal processing and power/driver chips), and is not relevant to playbacks. The interface between the main board and the playback is typically a low-end microprocessor sending and receiving digital communications messages in two directions using a protocol such as IIC (Inter Integrated Circuit™). Streaming digital audio is typically sent directly from the playback to the main board via a dedicated bus. The playback controller chip may physically reside within the playback hardware itself (”smart” playback), on a separate substrate (daughter board), or even reside on the main board (up-integrated).

Architectural requirements for Radio are: separation of algorithm layer, low-level drivers, and physical hardware with well-defined interfaces between them; encapsulation of the main functions (tasks); a well-defined interaction between the tasks and OS; flexible task configuration based on user-requested features of the radio; capability for independent testing of each task; and the ability to perform integrated testing.

Regardless of the location of the playback controller, the layering architecture should remain the same. A schematic view of the top-most architecture is shown in Figure 1.

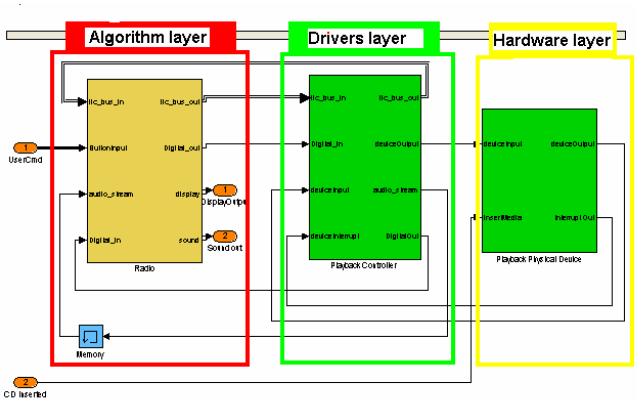

Figure 1. Radio Playback System Model architecture: Algorithm layer, Drivers Layer and Hardware layer.

Architectural requirements for each task are different from the Radio requirements. These requirements are: layered with well-defined interfaces; independent of operating system and development platform; capable of unit testing and regression tests; amenable to code reuse; and capable of managing states of operation. Principles of architectural development include: the use of contract programming (enforced interfaces) until just prior to production release; the use of graphical tools for facile communications; data and functional abstraction; and automated testing tools.

Playbacks execute typical commands known to all users of modern audio devices, such as: load, eject, play, stop/pause, scan, fast forward/reverse, seek up/down (track and/or folder), and options such as shuffle. The functionalities which are evoked in response to these commands are called behaviors. In event-driven radio applications it is possible to receive the request for execution of new behavior before the completion of the previous behavior. The priority and interaction of behaviors can be quite complex, and in many cases depends on specific customer requirements. Handling the execution sequence falls to a task “operation layer” which describes what is being commanded by the user or the OS. The behaviors themselves are collected in the task “functional layer”. Finally, a task “communication layer” handles two-way interface between the playback and the radio’s main board. A schematic view of the architecture is shown in Figure 2.

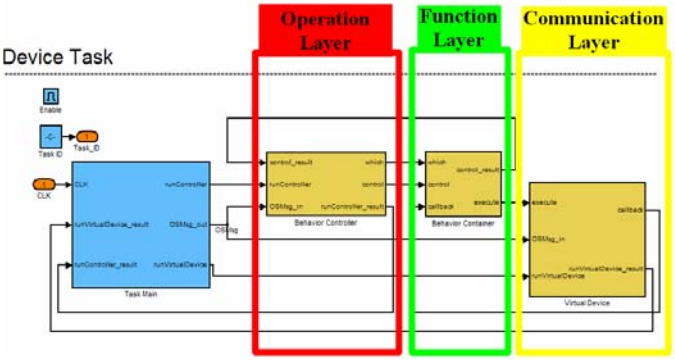

Figure 2. Playback Task follows a 3-layer architecture: Operation, Function, and Communication.

The design, modeling and simulation were performed using MATLAB/Simulink/Stateflow™ (The Mathworks, Inc.), which is widely used in automotive electronics system design. In order to comply with the architectural requirements and principals outlined above, we construct each behavior in the uniform way, consisting of three main blocks: Initialization, Run, and Finish, as shown in Figure 3.

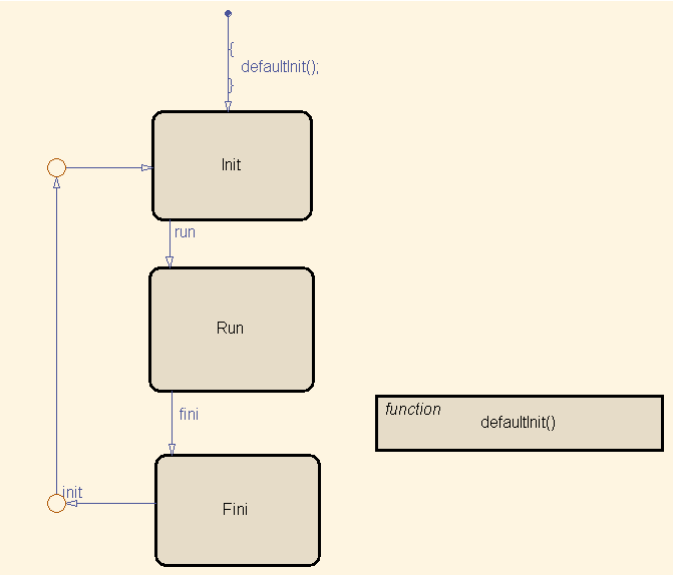

Figure 3. Behavior architecture, with Initialization, Run, and Finish steps. The "graphical function" defaultinit() is run upon entry into the behavior.

The behavior receives an event from the Behavior Controller, which is the main block of the Operation Layer in Figure 2. These events command the behavior to cycle through the Initialization, Run, and Finish states. Using these states allows us to clearly implement the functional abstraction and the concept of contract programming. User requests reach the Behavior Controller via pre-defined messages from the OS. The Behavior Controller assures that the current behavior completed its Finish state prior to initiating a newly-requested behavior. The Behavior Controller communicates through the Virtual Device layer which converts OS commands into IIC messages. The Virtual Device symmetrically translates and relays incoming IIC messages from the behavior to the Behavior Controller. Only after the Behavior controller receives the acknowledgement that the Finish state of the current behavior is completed will it switch to the new behavior and issue the Initiation event to that behavior.

ERROR HANDLING

If every CD followed Red Book standards and never received a scratch, if every playback worked perfectly, and if no driver allowed their children to insert hotel keycards into their car radios, playback control would be a straightforward task. The first phase of this model-based development project was to capture a generic set of requirements for a playback unit. It was discovered that specifications vary considerably among the vehicle manufacturers (VM). Each VM has a certain style and function set in their radios (infotainment systems), precluding standardization and complicating reusability. Each playback vendor has their own ideas on how to handle skips: some attempt recovery while others remain oblivious. The interaction between desired behavior and the countless ways in which a radio can be misused and abused make the handling of errors the most important feature of a playback controller. Developing an exhaustive set of requirements is believed to be intractable. Instead, we developed a unique strategy for error handling.

We adopted a hybrid error-handling strategy midway between competing extremes. One the one hand, a global error handling unit could be created with logic routines for every conceivable glitch. It would be memory intensive (and therefore costly) but universal; yet over the life of the vehicle its algorithms would become outdated, and require periodic upgrades. On the other hand, a small-sized error handling routine can be developed for each behavior. This is convenient for control engineers, but engenders considerable redundancy, since many playback Behaviors are affected by a given error. Our hybrid strategy abstracts error handling in each layer of playback tasks with the goal that error-handling is processed at the lowest level possible. If an error is discovered at the Communication Layer and can not be corrected at that layer, the information is "passed" to the next higher level of the architecture, the Function Layer. The Function layer tries to correct the problem, for example by repeatedly sending the command to the lower Communication layer. If the problem persists, then the error message is passed up to the Operation layer. The Behavior Controller attempts to fix the error, for example by re-initializing the behavior. Should the problem still be irrecoverable, a disc eject may be initiated. Should that fail, a signal is sent to the main board for final dispensation, and a message is passed to the HMI.

VALIDATION AND HARDWARE-IN-THE-LOOP

The standard practice of functional decomposition teaches us to break complex problems down hierarchically until we have "atomic" tasks, those which can be accomplished by an individual or small team without further input. Their output is a functional "unit", which can then be tested to derived requirements.

In traditional validation, software units which comprise playback control are integrated and tested on a lab bench. However, coding or logic errors within the units are difficult to debug once they have been downloaded to the controller. To address this issue, we have created virtual test environments for the radio main board (including the operating system), for the playback controller, and for the playback mechanism itself.

For unit testing of the Playback, we created adaptive virtual test fixtures which "wrap" a given behavior. Through this wrapper, we can apply test vectors (manually entered or created automatically through scripts) and capture the outputs. These wrappers work for each level of the architecture, allowing us to perform unit test and integration tests in a virtual environment.

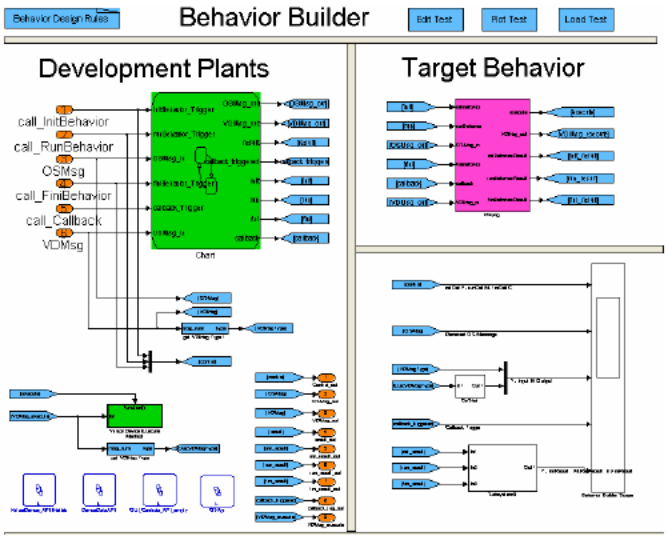

Figure 4. Screen shot of Simulink adaptive test fixture. The behavior under test inserted into this standard template is the large magenta rectangle at right-center.

Rapid, virtual execution of unit tests provides a three-way improvement in validation: more tests can be performed within a given time constraint, with the potential to discovery more bugs; regression testing is highly automated, accelerating future design turns; and, software can be tested in advance of availability of production-intent hardware (such as microprocessors).

Integration tests provide the same improvements in validation as unit tests. Yet to be relevant, the virtual integration test environment must map readily to a hardware-in-the-loop (HIL) environment.

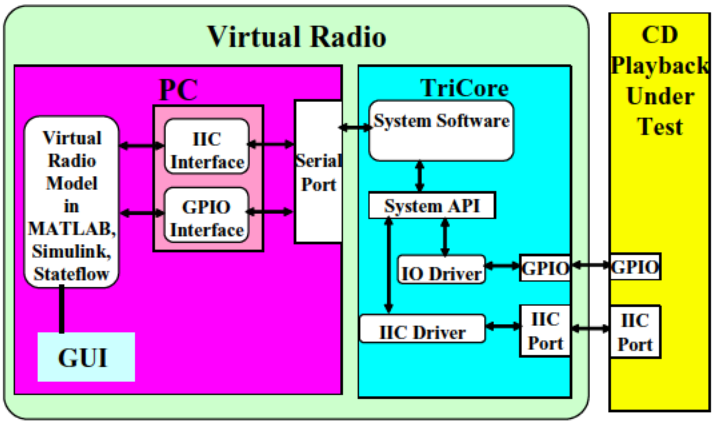

Figure 5. Schematic of hardware-in-the-loop setup.

Figure 5 shows a schematic of the "virtual radio" wherein any major component can be either hardware or a software model. The graphical user interface (GUI) was created using The Mathworks' GUIDE tool, superposed on a bitmap photograph of a generic radio faceplate, as seen in Figure 6.

Figure 6. Radio GUI for hardware and for software.

In the assembly of the HIL lab bench, we discovered a lack of tools to perform IIC master operations, which were required for the target playback. We purchased a TriCore™ (Infineon Technologies) evaluation board, adding configuration and software to perform as both an IIC master and IIC slave between the playback and the radio. The TriCore has a configurable number of inputs and outputs (IO), allowing it to be adapted readily to a variety of playback mechanisms.

Within the virtual radio, we modeled the operating system (OS) as a Java™ (Sun) executable, running in the background of the MATLAB process. The OS was hand-written for this project, based loosely on the MicroOS™ (Digital Corporation) specifications.

AUTOCODE

The logic for behaviors, error behavior and communications is captured in Stateflow, a powerful tool for modeling of event-driven systems. Interfaces and control signals are captured in Simulink. Data processing and host PC interface is done in MATLAB using M-scripts called by the Simulink model or by the OS Java executable. For this case study of an audio playback, only the Stateflow algorithms were converted to C code suitable for compiling and download to a microprocessor.

Upon completion of unit testing and HIL testing, we used the TargetLink™ tool (dSPACE) to automatically generate ANSI compliant C-code. A data dictionary, developed in-house at Delphi, was employed to capture engineering variable characteristics and to map software variables to hardware IO. TargetLink provides automated documentation in html format which facilitates reading and review of the generated code.

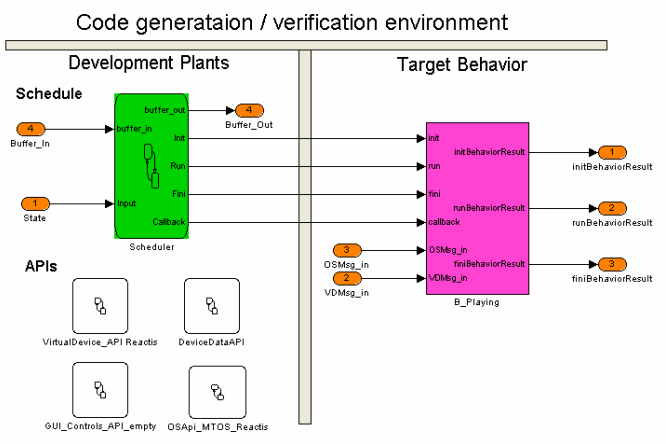

Figure 7. Screen shot of Code generation / verification environment. The behavior under code generation or test inserted into this standard template is the large magenta rectangle at right.

The environment for code generation and software unit tests for every Behavior is similar to the environment for the model's unit testing. The same functional test vectors, which were use in model's unit tests, were re-used for verification of generated code against the model. In addition to functional test vectors, the suite of model-coverage based test vectors were generated by employing the automatic test vector generation tool Reactis™ (Reactive Systems). A comparator function, which is part of the TargetLink code generation tool, captured any discrepancies between the model behavior and the behavior of auto generated code. When all discrepancies are identified and resolved, and all tests pass, the units are released. Finally, the generated files were linked with hand-implemented OS and HWIO drivers to run on hardware. A comparison between autocoded software development times and manually-generated playback control software is shown in Table 1.

	Coding (hours)	Testing (hours)	TOTAL (hours)	TOTAL (bytes)
Hand-code	Not available	Not available	Not available	14903
Auto-code	6	8	14	12437

Table 1. Comparison of development times in hours and bytes for hand-code versus autocode (including Stateflow modeling).

LESSONS LEARNED

Near the project endpoint, a new playback unit was being considered. As a test case, we adapted the interfaces to the new unit in only four hours. This demonstrates the considerable time savings possible with a modular architecture, reusable building blocks and customizable, adaptive interfaces. Production-intent code was validated virtually, on the lab bench, and has undergone a full peer review per corporate procedures.

LIMITATIONS

Software developers accustomed to the C language expect pointers, which MATLAB lacks. The long list of tools used in this work is obviously a detriment to training. Coding the OS in Java was our second choice, since modeling a pre-emptive OS in MATLAB is difficult. Capturing communications logic in Stateflow is not suited to the lowest level of detail ("bit-banging"), and is not "cycle-accurate", so that very low level operations do not gain from its use. We realized that only a very well-defined development environment and process would allow efficient use of MATLAB/Simulink/Stateflow for production event-driven applications. Creation of such an environment is not a trivial task and requires advanced knowledge of modeling tools, and sufficient time away from the typical pressure of production schedules.

BENEFITS

We found that the visual nature of the architecture made it accessible to high-level managers and less-technical supervisors, as well as facilitated technical discussions amongst our team. We found legacy code easy to integrate with either Stateflow or TargetLink. Code was generated and tested just in fraction of time that is typically required for hand-code implementation of the same algorithm. Our test bench works for unit testing, integration testing, and can even work with the entire radio. Re-use is quick and convenient, as evidenced by our 4 hour swap-over. The use of HIL allowed us to validate not only the developed algorithm, but the playback mechanism from new supplier. We discovered the features of this mechanism, that were not described in the supplied documentation, and adjusted our control algorithm accordingly.

CONCLUSIONS

Using a suite of engineering automation tools centered around MATLAB, we have established a playback architecture with distinct building blocks and well-defined interfaces. This architecture supports re-usable building blocks drawn from on-line libraries. The simulation environment created allows testability of units and integration within the playback functionality. We are able to develop and evaluate control algorithms which operate in a virtual environment, and we have created the capability to run these tests on the playback device in a HIL configuration. From the behaviors and error handling implemented in the model environment, we have generated autocode, compiled this on the host, and used it to communicate and operate a playback mechanism through the HIL interface. Throughout this exercise, we have learned to more clearly identify where the tools and methodologies are helpful in product

development, as well as learning their limitations so that we can apply them where the greatest benefit can be obtained. A number of auxiliary tasks were completed that were not originally anticipated, but which need be performed only once. The use of model-based development provides a platform which facilitates rapid turn-around on playback control development.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge participation on this project by Greg Kuhlman, Gary Mitchell, Doug Srail, BC Manjunath, Jim Costa, Jason Molenda, Kirk Bailey, Jill Hersberger and Becky Cobb. We also extend our thanks to the champions and sponsors of this project: Randy Brunts, Jeff Jones and Michael Neuhalfen.

REFERENCES

1. "Model Based Systems Development in Automotive," M. Mutz, M. Huhn, U. Goltz, C. Kromke, SAE World Congress 2002, paper 03B-128.
2. "Model-Based Tools Update," The Hansen Report on Automotive Electronics, June 2001, vol. 14, no. 5.
3. "Incorporating Autocode Technology into Software Development Process", L.Vitkin, T.K.Jestin, ICSE 2004, pp.51-57.
4. "Managing the Challenges of Automotive Embedded Software Development Using Model-Base Methods for Design and Specification", M Yeaton, SAE 2004-01-0720
5. "Automotive Software development: A model Based Approach", M.Rappl, P. Braun, M.von der Beek, C. Schroder, SAE 2002-01-0875

Contact information:
Peter J. Schubert, Ph.D.
Packer Engineering, Inc.
1950 N. Washington St.
Naperville, IL 60566-0353
800-323-0114
pschubert@packereng.com